- **三个放弃** 指令/代码执行原子性假设不再成立
- 程序的顺序执行假设不再成立 多处理器间内存访问无法即时可见
- "处理器一次执行一条指令"的基本假设在今天的计算机系统 上不再成立(我们的模型作出了简化的假设)。
- 状态不再仅仅由自己改变,由于共享内存

几个结合后的难处

- 不同体系的存储结构是不一样的,可能是x86,也可能是risc的分布式。

正确的Peterson算法。

目前来看,CPU的硬件优化,编译器的编译优化,存储系统的分布式和OS的并发相关内容产生了奇妙的联系。

能失效,在现代的处理器中,你需要使用他们的协作来实现对 应的功能。)

A和B争用厕所的包厢

- 想讲入包厢之前。A/B 都首先举起自己的旗子

- 出包厢后,放下自己的旗子(完全不管门上的标签)

- ト 無江立 のだけ中級 ・ 成者 B 想进 但还设来得及把"A 正在使用"贴在门上 名 看到 B 幸議子 ・ 人一定已起往陽子举起来了 ・ (1億~46.1%~(6.1%%)

- 绕来绕去很容易有错漏的情况

Prove by brute-force

- 枚举状态机的全部状态(PC), PC), z, v, turn) 但手写还是很容易错啊——可执行的状态

- 日の推加の名の立「内付か大陸 「知知機能の協議を終う権 ・ 3回域を配付上が下参照時、 業計正道時時 ・ 6度下海子之際 ・ 6度下海子之際 ・ 1世の大学で、1987年 ・ 1987年 ・ 1987

互斥(原子指令的属性) 根据上述的并发模型我们可以知道,lock和unlock天生自带阻止编译优化。

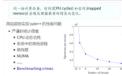
- - 原子的操作,无法被打断的

自旋锁,scalability,优化

多线程,如果对于锁的争抢变多,反而不如单线程。(通信的开销。)

(唯一的) 使用场景:操作系统的 并发数据结构。(OS内部很难避免。)

Scalability: 性能的新维度



关于互斥的一些分析

- 更快的 fast path
 xchg 成功 → 立即进入临界区、开销很小

分区 jyy的OS 的第1页

- 更慢的 slow path
 xchg 失败 → 液费 CPU 自旋等符

互斥锁 (通过系统调用访问 locked)

- 编译器假设县顺序执行。不会有其他人来修改你的内存。所以会做出一些在单处理器时正确的操作。这个
- CPU类似于动态编译器,这个导致我们有时候很难预测他的行为

Peterson 算法在现代,不能简单的照搬,不然会失效。我们需要处理器和编译器的协作,才能写出现代意义上

人类天生的难以理解并发。

许多个处理器做到动作,然后做一个全局的排序,这个是NP完全问题,指数级时间,

并行的前提是、计算远大于通信的时间。

Peterson 算法简介

他的假设前提是,读和存是原子的。 (所以在现代处理器中可

- A 往厕所门上贴上 "B 正在使用" 的标签
- B 往厕所门上贴上 "A 正在使用" 的标签
- 然后,如果对方举着旗,且门上的名字是对方,等待
 否则可以进入包厢

如何验证peterson算法的正确性?(关于自动化验证)

你可以写个程序。来帮你证明对应的操作是正确的。

- 习题: 证明 Peterson 算法正确,或给出反例
- 如果只有一个人举旗、他就可以直接进入
 如果两个人同时举旗、由则所门上的标签决定指
 手快 (被另一个人的标签覆值)、手惯
- 非自体的細节機の

- void TA() { while (1) {
 /* @ */ x = 1;
 /* @ */ turn = B;
 /* @ */ while (y && turn == B) ;
 /* @ */ x = 0;] }
- void TB() { while (1) {
 /* ① */ y = 1;
 /* ② */ turn = A;
 /* ② */ while (x && turn == A);
 /* ② */ while (x && turn == A);

自动遍历状态空间的乐趣

- 都特施度(状态空间)上的通历问题了!

自旋锁,低效率。

但是,CPU关中断不能关别人的,自旋是没有办法避免的。 采用了两端锁和sleep来尽量减少开销。

- 现实中的并发编程 高性能计算,数值密集型科学计算任务
 - 计算任务如何分解? (生产者消费者解决一切。)
 - 海量线程之间如何同步和通信。

(也就是,解决两个问题, 1, 计算图是什么2, 如何分解任务, 如何同步通信。)

数据中心【数据存储为中心(互联网搜索,社交网络,联网应用)】 特点:事件驱动+高并发:系统调用密集且延迟不确定

由哲学家吃饭引出的一些思考

吃饭是,分布式的,自组织的形式。

来个服务员,就类似于OS的调度。

- 读取网络包
- 在服务海量地理分布请求的前提下,三者不可兼得。数据一致 (Consistency)
- 服务时刻保持可用 (Availability) 容忍机器离线 (Partition tolerance)
- 。 单机程序目标,尽可能多的服务并行的请求
 - QPS: 吞吐量 Tail latency: 一个请求慢了, 其他的请求不能慢。

因此,线程不够用了,他能并行但是很慢,远多于处理器的线程会导致性能问题。而且切换开 绺 维护开绺很大 6月77月中了协程

例子: 求和)男个様職、計算1+1+1+...+1(部計2m个1) Rdefine N 100000000

- 119790390.99872322 (結果可以比 N 还要小)....

编译器的不同优化,也会对程序造成不同的影响。

02和01,02对了,01没对。

编译器的求和,可能会直接把一个等差数列。直接化成一个对应的常数,然后这个就可能会导致一些在并发过程中错误。 所以这个错误是偶尔会出现,我也不可能不会出现,你不能简单的认为 02的出错,一定会比欧一多。

O2直接把循环展开成常数了=_=

编译器假设你的代码是是顺序执行的,果他不假设你顺序执行指令,那 么他就基本没有办法优化任何东西。

如何保证编译正确性? 插入barrier,不可优化的代码。标记变量为不可优化,volatile。

你哪怕只有一条指令,他可能会被并行弄掉。 前哪回不同一来指令,他问能去放开门开拜。 就算是只有一条指令,它的执行也可能不是原子的,比如说x86。指令并 不是原子的,除非它指定了使原子指令。一条指令似乎也不是原子的? 还是一种性问题? 指令的执行,并不是原子性的。(世界观崩塌了,我 不知道是同时写还是微指令的问题。)

<mark>计算图与状态机</mark> 最长公因子,动态规划,etc.

你可以为他们画出一个计算图,然后对这个图进行拓扑排序,就能能得出对应的计算顺序。

图的每个顶点都代表一些计算。 每个计算图,都有一个最长的路径,这个路径需要花费的时间,就是你不论怎么并行都没有办法 优化的时间,所以最长路径就是你要花的,至少的时间。

各种各样的图。状态机,前后依赖,这个在CS里面是很常见的。 体系结构,现代很重要的一个点就是,处理的时候,用我们的机器做乱序的指令级别排序。 (用硬件实现拓扑排序)

打印心用信号量会极其。。。。所以基本不讲信号量,用状态机。 列出对应的状态机,然后对着状态写转移代码。 条件变量是万能的.

万能的并行框架 这段代码的执行条件是什么,不管3721,先上锁,然后检查条件,不满足就释放,sleep。 然后唤醒所有人 (广播)

信号量(条件变量的特例)【不应该是条件变量的泛化,可

以是锁的泛化,拓展的互斥锁】

- 避免broadcast的浪费 写出优雅的代码
- 使用场景 happens-before (临时性的) 计数型的同步 (拥有多少资源,顺序不影响)
 - 信号量的两种典型应用
 - 初始: s=0
 - A; V(s)
 - P(s): B 假设 s 只被使用一次、保证 A happens-before B
 - 2. 实现计数型的同步

 · 初始: done = 0

 · Tworker: V(done)

 Tmain: P(done) × T
 - 对应了两种线程 join 的方法 • $T_1 \rightarrow T_2 \rightarrow \dots$ v.s. 完成就行,不管顺序

分布式,异步,不同的程序异步性,同步本身是很困难的**。**

用尽可能简单的机制,去实现现实中存在的一个协议。

来个服务员会不会好很多?同时也避免了死锁,分布和集中。

使用信号量实现条件变量: 本质困难

例子: 实现计算图

- 対等条入法地行P(wait)操作
 充成计算任务
 対等条出边执行V(post/signa
 等条边抢好P一次、V一次
- 每条边恰好P一次、V PLCS 直接就解决了啊
- 有先后顺序关系,是happens-before! tip: 为了降低通信的一些开销,你划分的力度可能会变得粗一点。
- 比如ace划分到一起,然后就只需要3次通信。 实现计算图 (cont'd)
- 作一看很厉害 完美解决了并行问题

协程:操作系统"不感知"的上下文切换

世一直的行,直到 yield() 主观原外处理简 • 有编译器辅助、切换开册低 · yield() 是函数调用,只需保存/恢复 "callee saved" 寄存器 · 线程切换需要保存/恢复全部寄存器

Go: 小孩子才做选择,多处理器并行和检查级并发我全部要!

routine:概念上是线程,实际是线程和协程的混合体

每个CPU上有一个Go Worker,自由调度 goroutines
 执行到 blocking API 时 (例如 sleep, read)

• 但"一直执行"、直到 vield() 主动放弃处理器

但等待 I/O 时,其他协程就不能运行了……
 失去了并行

// 尺寸能差 1122 点 2211 void T1() { send("1"); send("1"); yield(); } void T2() { send("2"); send("2"); yield(); }

和线程概念相同(独立地栈、共享内存)

Go 和 Goroutine

・同期MACU TO TO TO THE LINE DOE

・ 回転形点 SCHEPLING S 信号量 = Time Line Doe

・ 配配性放大 SCHIPME
・ 一个系数例を予ける点的計算
・ 中心系数例を一次定因了五戸者・持奏者
・ 現成代象上式即使用 ・ 可能出版「物の数」・ 又及因了五戸者・持奏者
・ 可能出版「物の数」・ 又及因了条件交量

現代处理器也是(动态)编译器!

- 一个CPU执行一条划合到达下一状态
- RF[9] = load[RF[7] + 4
 store[RF[12], RF[13])
 RF[3] = RF[4] + RF[5]
- 与编译器一样、做"顺字执行"假设:没有其他处理器
 每一周期执行尽可能多的 Aop 多路发射、乱字执行。

放弃(3):多处理器间内存访问的即时可见性

满尺半处理器 eventual memory consistency 的执行,在多处理器系统上可能无法序列化!

- 当ェナッ町、対エッ的内存读写可以交換順序
- 它们甚至可以在同一个侧断里完成 (只要 bad/store unit 支持)
 如果写 z 发生 cache miss. 可以让误 y 先执行
 满足 "尽可能执行 pop" 的原则。最大化处理器性能

在多处理器上的表现 两个处理器分别看到y=0和x=0 乱序多发射(超标量提升效率)+ 重排序缓存(对

外假装顺序执行) 假设我对x和y两个变量进行操作,如果这两无关,

我完全可以认为,执行的顺序是没有什么关系的。

这也是 Intel 自己给自己标的包袱
 看看 ARM/RISC-V 吧。根本就是个分布式系统

x86,所有的写都是写进一个队伍里的,这个就导致,他 们不同线程读取到的值,如果同时读取,读取到的东西起 码是一致的。但是在RISC,就很难说。 宽松内存模型, 处理器难以维持准确翻译的一个很重要的



实现互斥: 做题家 V.S. 科学家

- 如果长久的训练都是"必须在规定的时 么浪费时间的图号自然就少了

计算密集部分(1): SIMT

执行流有独立的寄存器 • z, y, z 三个寄存器用于标记"线程号",决定线程执

多个cpu共同用一个pc. 只要线程并行的足够多. 我 们就可以很好的利用这个特点,但是处理那种条件判



计算密集部分(3): 堆更多的处理单元!

断会非常的慢。

例子 i int x = 0, y = 0; void T1() {
 x = 1; // Store
 __sync_synchroni
 printf("Kd", y);
}

void 12() (

- 適历模型告诉我们: 01,10,11

明的想一点东西,

• 机器永远是对的 Model checker 的结果和实际的结果不同→假设错了

完全不符合编程模型的例子 事实上,会出现00(而且这个的出现情况次数远远大于 其他,11很少出现) 现代的CPU和硬件极其复杂,所以不要让自己显得很聪

现代处理器实际上是动态编译器。维护一个微操作的池

子,而且做顺序执行的假设。 宏松内右續型 (Relaxed/Weak Memory Model)

- 宽松内存模型的目的范徒早处理器的换行更高效。
- 86 已经是市面上邮买到的"最强"的内存模型了 👄



(x86-TSO in Hard 宽松内存模型

无锁算法

veid lock(lock_t *lk);
veid unlock(lock_t *lk);

科学家、考虑更多更被土的问题 · 我们可以设计也多样的原子指令? · 它们可能战场力的阿子 · 计算知题中以现此比一次 lead/viore" 图像的原子性唉 · 如果提杆模因使、软件/编译器可以点?

- 一个PC. 控制32个执行流同时执行
 逻辑线程可以更多

计算密集部分(2): SIMD



• 144 SMs • 18432 CUDA Cores (并行的 Thread • AVX512: 512bits = 16 x Float32

Compare and exchange (test and set) 更加强大的原子指令。

- xchg 成功 → 立即进入临界区、开销很小
- 東便的 skow path
 xchg 失败 → 浪费 CPU 自旋等符

互斥锁 (通过系统调用访问 locked)

- 更经济的 slow path
 上锁失败线程不再占用 CPU
- 更慢的 fast path
 即便上號成功也需要进出內核 (syscall)

- Fast path: 一条原子指令,上驱成功立即返回 Slow path: 上坡失致,执行系统调用暗采 性能优化的最级现取了 。 相 average [frequent] case 而不是 worst case
- POSIX 线程库中的互压镜 (pthread mutes)
- 观察线程库中的 lock/unlock 行为
 1. Mutex 没有争抢的情况
 2. Mutex 看争抢的情况
- QPS: 吞吐量
 Tail latency: 一个请求慢了,其他的请求不能慢。 因此,线程不够用了,他能并行但是很慢,远多于处理器的线程会导致性能问题。而且切换开

销,维护开销很大。所以引出了协程。 传统的语言机制也许不是很适合并发,所以出现了go。

- AI模型 (计算密集 + 数据密集)

■ 谷忍机器電线 (Partition tolerance)

- 浏览器

有模型。就可以模拟。很多任务。且有空间局部性。天然就是并行的。可以不断地分成小部 分。物理世界,交互出现在局部性。

G0 ≱µ Goroutine

Go: 小孩子才做选择,多处理器并行和检查级并发我全部要!

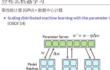
Goroutine: 概念上是线程,实际是线程和协程的混合体

- ・ 等へで以上等へのWarterを設定。 ・ 第行列 blocking API 时 (例如 decept read) ・ のの では、 ・ では、 ・

Go 语言中的同步

Do not communicate by sharing memory; instead, share memory by communicating. ——Effective Go

- 共享内存。万忌之道 信号量/条件变量: 实现了同步,但没有实现"通信" 数据传递完全靠手工(设上铁铁指了)
- 但UNIX时代就有一个实现并行的机制了
- cat *.txt | wc -l
 普遍是一个天然的生产者/消费者!
 为什么不用"管道" 实现协程/线程间的同步 * 通信呢?
 Channels in Go
- 传统代码都是共享内存,但是实际上不能实现传递信号的作用。



显存/缓存决定了GPU内堆处理器的上限但我们可以有多台机器、每个机器有多台GPU!

计算密集部分(3): 堆更多的处理单元!

汶只是一个 GPU

分布式机器学习

如何解决死锁? (事实上,面对死锁,我们在代码层面基本只能使用上锁顺序来解决。前三个假设都是破坏掉了互斥的假设)

- Mutual-exclusion Wait-for
- No-preemption (非抢占) Circular-chain

数据竞争(data race)(since C11 Undefined behavior) 【不同的线程同时访问同一内存,且至少有一个是写】

- Weak memory model 在实际中,最常见出现的就是,
- 1. 上错了锁
- 2. 忘记上锁。

"内存" 可以是地址空间中的任何内存

- 可以是全部变量
- 可以是堆区分配的变量
- 可以是栈

"访问" 可以是任何代码

- 可能发生在你的代码里
- 可以发生在框架代码里
- 可能是一行你没有读到时的汇编代码
- 可能时一条 ret 指令

- 原子性被破坏 (ABA) 【互斥锁】
- 顺序被破坏 (BA) 【条件变量/信号量】

人举是sequential creature.

并发机制后果自负。

但凡是你在online judge里面调了很久的,都是一个愚蠢的问题,比如你把0写成了1, 那种看上去你绝对不会出错的地方。

函数式编程, 尽可能让你的函数没有副作用, 让你的代码易干维护。

如何应对并发bug

解法全局编号,给你要上的锁弄个编号,凡是要获取锁的话,必须先获取小的锁,再获取大的 锁。那么如何获得对应编号?

可能是按照每一行的锁,给对应的锁编号。实现应该很多,因为上锁是需要进行系统调用的,可 以比较好的进行。

- 数据竞争

访问是共享内存,和系统调用无关,用编译器,生成对应的图,看是否可达,来判断是否可能会 产生数据竞争。如果上锁的时候,能够到达其他修改这个变量的操作,就是有数据竞争。

编程语言是一种有损的转换。 (需求到具体实现的有损。) bugs,违反程序员对物理世界的假设和约束。

编程语言的specification

编程语言与 Bugs

- 编译器/编程语言 只管"翻译"代码,不管和实际需求(规约) 是否匹配
- "山寨支付宝"中的余額 balance
 正常人看到 0 ~ 18446744073709551516 部认为"这件事不过"(balance" 自附 no-underflow 的企义)
- 怎么才能编写出"正确"(符合 specification) 的程序?

- ・ 正明: Annotation verifier (Dafny), Refinement types
 ・ 推測: Specification mining (Daison)
 ・ 投流: Program sketching
 ・ 構理語言的历史如果果
 ・ り掲載者
 ・ り掲載者
 ・ 「略音音・承報者書

防御性编程,做好测试,所有你觉得,可能不对的情况都检查一遍。assert的使用。 例子:peterson临界计数器,二叉树旋转,AA型deadblock检查。

#define 定义宏,根据变量的含义, 检查对应的变量。

所有的东西,最终犯错的还是人。一个已近严重到,而且普遍到成为某种现象,而且有固定模型 的bug,那么一定会有人能判断出来。

如何解决lock ordering? 就是如果写出了不对劲的lock order,就拒绝编译。

Lockdep 数据竞争

- Thread sanitizer(同样还有各种的sanitizer,address,memory,UB)如果你能得到对应日志,剩下的就是一个简单的数据结构题了。

你的工具可能比较简陋,但是必须要有,简陋,某些不大合理的前提下,很有用,很简单。

可以误报,漏报。 (你可以count—亿次后就panic) 金丝雀,填充magic number,在切换的时候进行检查 kalloc 简易判断自己内存分配。 手搓小的debug工具。

现实中会存在不可被屏蔽的中断。